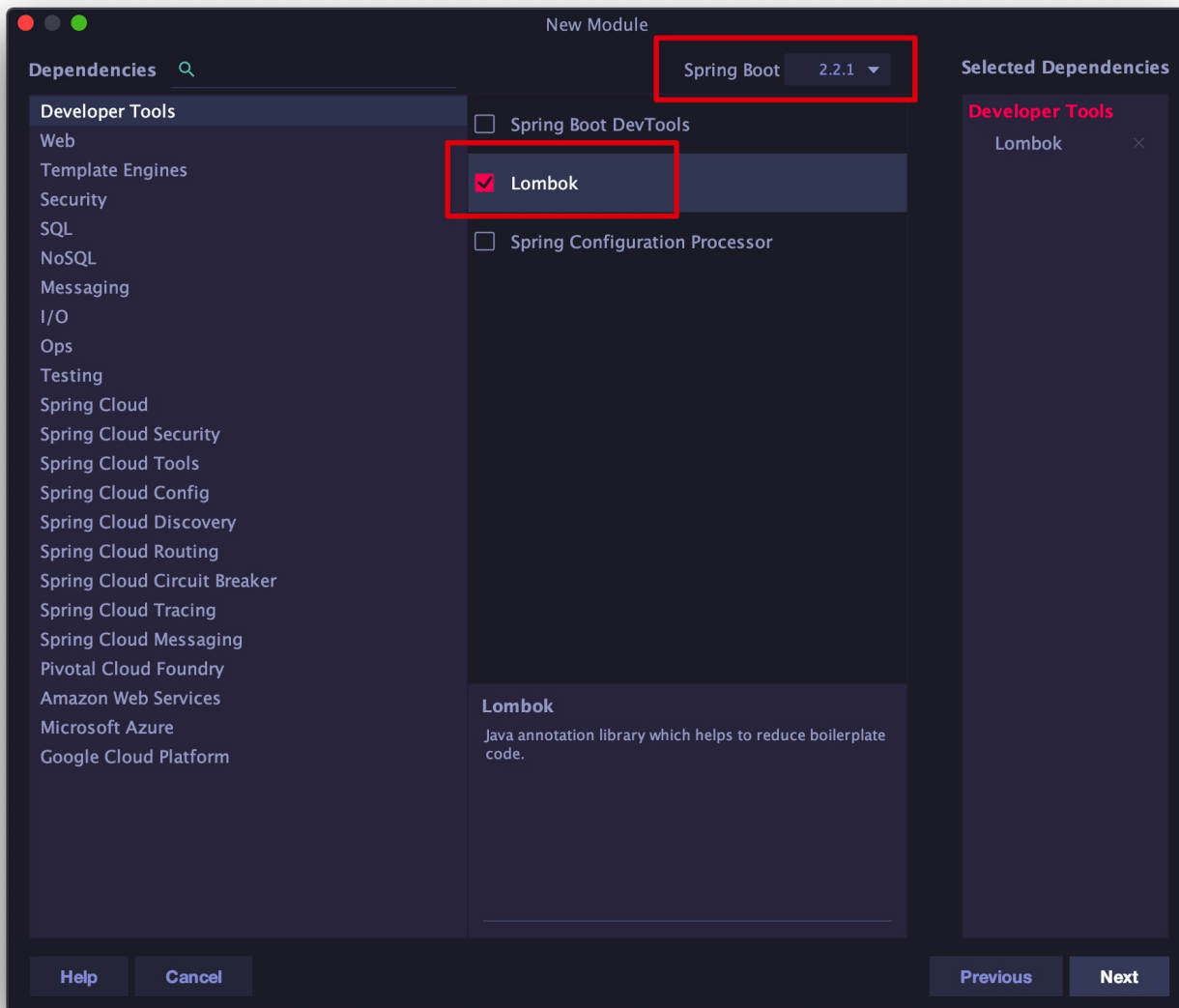


前言

在 Java 应用程序中存在许多重复相似的、生成之后几乎不对其做更改的代码，但是我们还不得不花费很多精力编写它们来满足 Java 的编译需求

比如，在 Java 应用程序开发中，我们几乎要为所有 Bean 的成员变量添加 `get()` ,`set()` 等方法，这些相对固定但又不得不编写的代码浪费程序员很多精力，同时让类内容看着更杂乱，我们希望将有限的精力关注在更重要的地方。

Lombok 已经诞生很久了，甚至在 Spring Boot Initializr 中都已加入了 Lombok 选项，



这里我们将 Lombok 做一下详细说明:

Lombok

官网的介绍: Project Lombok is a java library that automatically plugs into your editor and build tools, spicing up your java. Never write another getter or equals method again. Early access to future java features such as `val`, and much more.

直白的说: Lombok 是一种 Java™ 实用工具，可用来帮助开发人员消除 Java 的冗长，尤其是对于简单的 Java 对象（POJO）。它通过注解实现这一目的，且看:

传统的 POJO 类是这样的

The screenshot displays an IDE with two panes. The left pane shows the 'Employee' class structure, listing methods and fields. The right pane shows the corresponding Java code for the 'Employee' class.

Left Pane (Employee Class Structure):

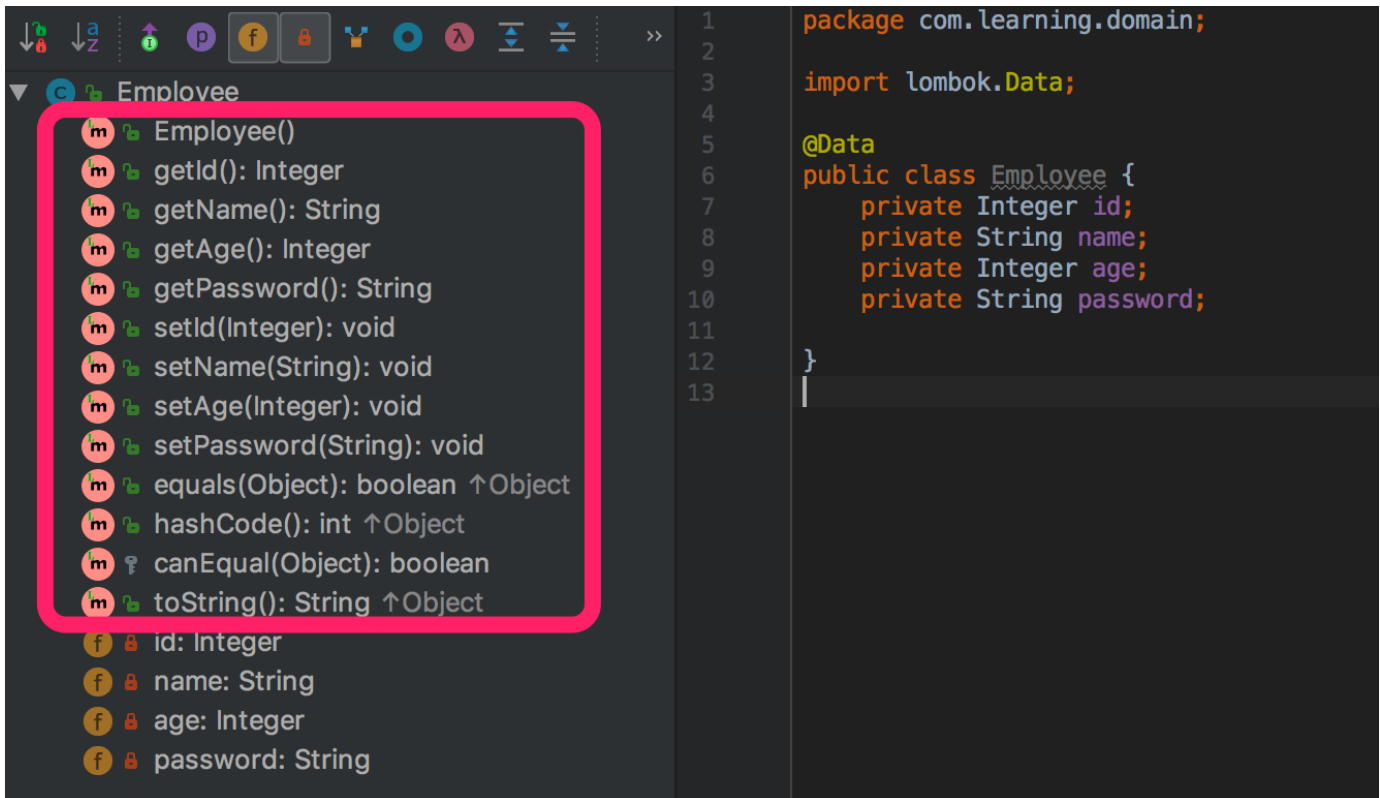
- Methods (highlighted in a red box):
 - `getId(): Integer`
 - `setId(Integer): void`
 - `getName(): String`
 - `setName(String): void`
 - `getAge(): Integer`
 - `setAge(Integer): void`
 - `getPassword(): String`
 - `setPassword(String): void`
- Fields:
 - `id: Integer`
 - `name: String`
 - `age: Integer`
 - `password: String`

Right Pane (Employee Class Code):

```
1 package com.learning.domain;
2
3 public class Employee {
4     private Integer id;
5     private String name;
6     private Integer age;
7     private String password;
8
9
10    public Integer getId() {
11        return id;
12    }
13
14    public void setId(Integer id) {
15        this.id = id;
16    }
17
18    public String getName() {
19        return name;
20    }
21
22    public void setName(String name) {
23        this.name = name;
24    }
25
26    public Integer getAge() {
27        return age;
28    }
29
30    public void setAge(Integer age) {
31        this.age = age;
32    }
33
34    public String getPassword() {
35        return password;
36    }
37
38    public void setPassword(String password) {
39        this.password = password;
40    }
41 }
42
```

The code on the right pane is enclosed in a red box, highlighting the getters and setters.

通过Lombok改造后的 POJO 类是这样的



```
package com.learning.domain;

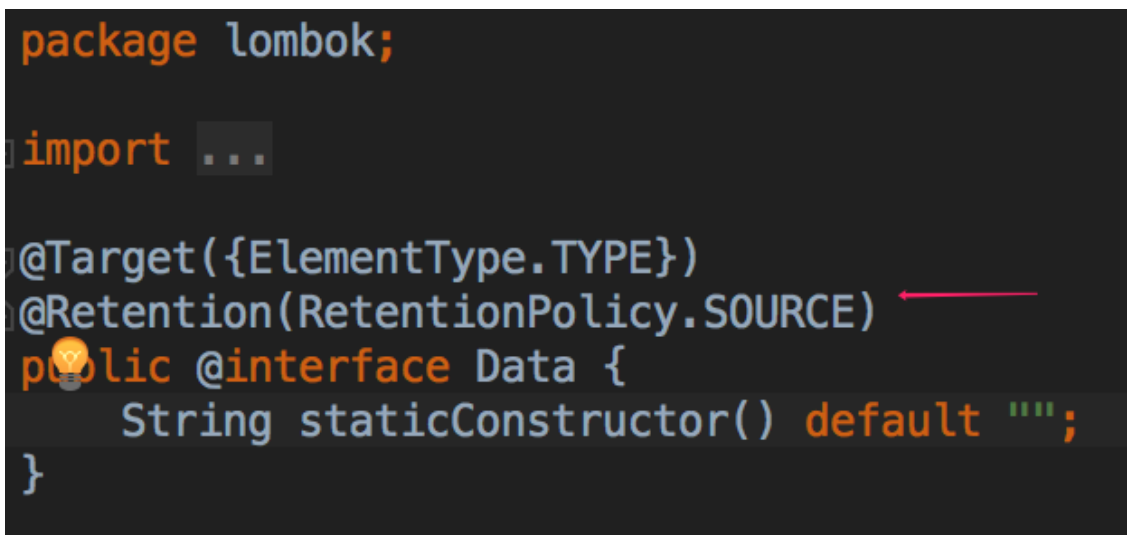
import lombok.Data;

@Data
public class Employee {
    private Integer id;
    private String name;
    private Integer age;
    private String password;
}
```

一眼可以观察出来我们在编写 Employee 这个类的时候通过 `@Data` 注解就已经实现了所有成员变量的 `get()` 与 `set()` 方法等，同时 Employee 类看起来更加清晰简洁。Lombok 的神奇之处不止这些，丰富的注解满足了我们开发的多数需求。

Lombok的安装

查看下图，@Data的实现，我们发现这个注解是应用在编译阶段的



```
package lombok;

import ...

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.SOURCE)
public @interface Data {
    String staticConstructor() default "";
}
```

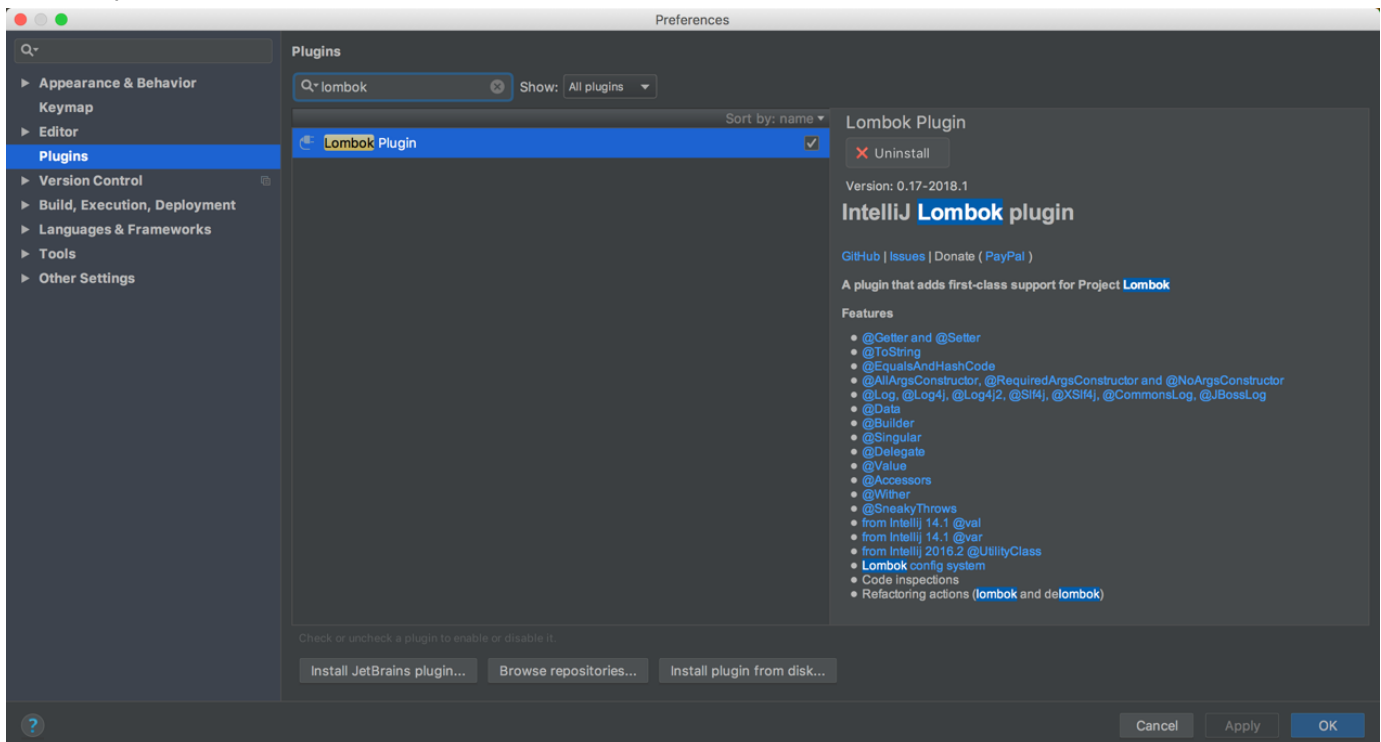
这和我们大多数使用的注解，如 Spring 的注解（在运行时，通过反射来实现业务逻辑）是有很大的差别的，如Spring 的@RestController 注解

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME) ←
@Documented
@Controller
@ResponseBody
public @interface RestController {

    /**
     * The value may indicate a suggestion for a logical component name,
     * to be turned into a Spring bean in case of an autodetected component.
     * @return the suggested component name, if any
     * @since 4.0.1
     */
    String value() default "";
```

一个更直接的体现就是，普通的包在引用之后一般的 IDE 都能够自动识别语法，但是 Lombok 的这些注解，一般的 IDE 都无法自动识别，因此如果要使用 Lombok 的话还需要配合安装相应的插件来支持 IDE 的编译，防止IDE 的自动检查报错，下面以 IntelliJ IDEA 举例安装插件。

在Repositories中搜索Lombok，安装后重启IDE即可



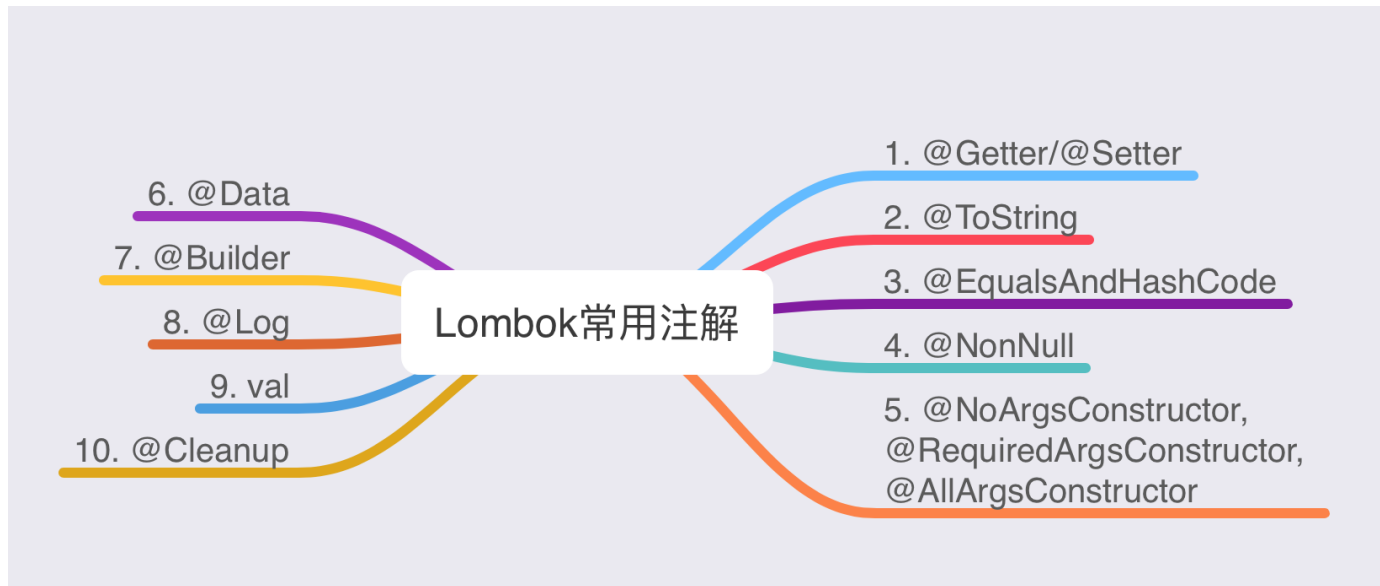
在Maven或Gradle工程中添加依赖

```
<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.10</version>
  <scope>provided</scope>
</dependency>
```

至此我们就可以应用 Lombok 提供的注解干些事情了。

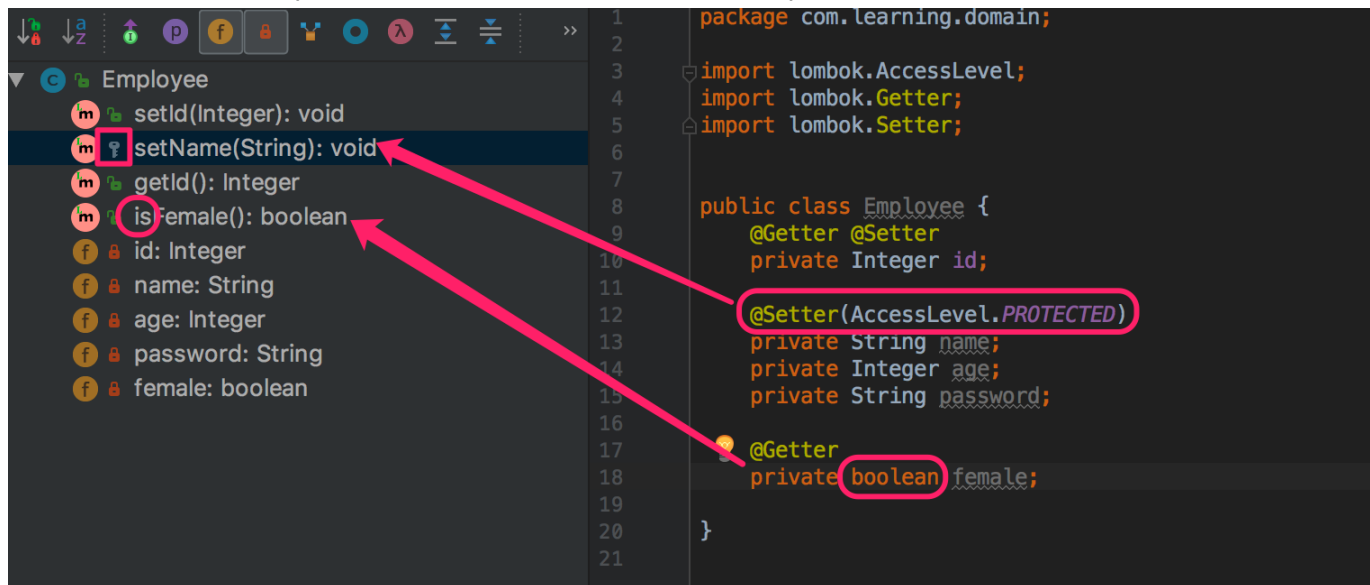
Lombok注解详解

Lombok官网提供了许多注解，但是 “劲酒虽好，可不要贪杯哦”，接下来逐一讲解官网推荐使用的注解(有些注解和原有Java编写方式没太大差别的也没有在此处列举，如@ Synchronized等)

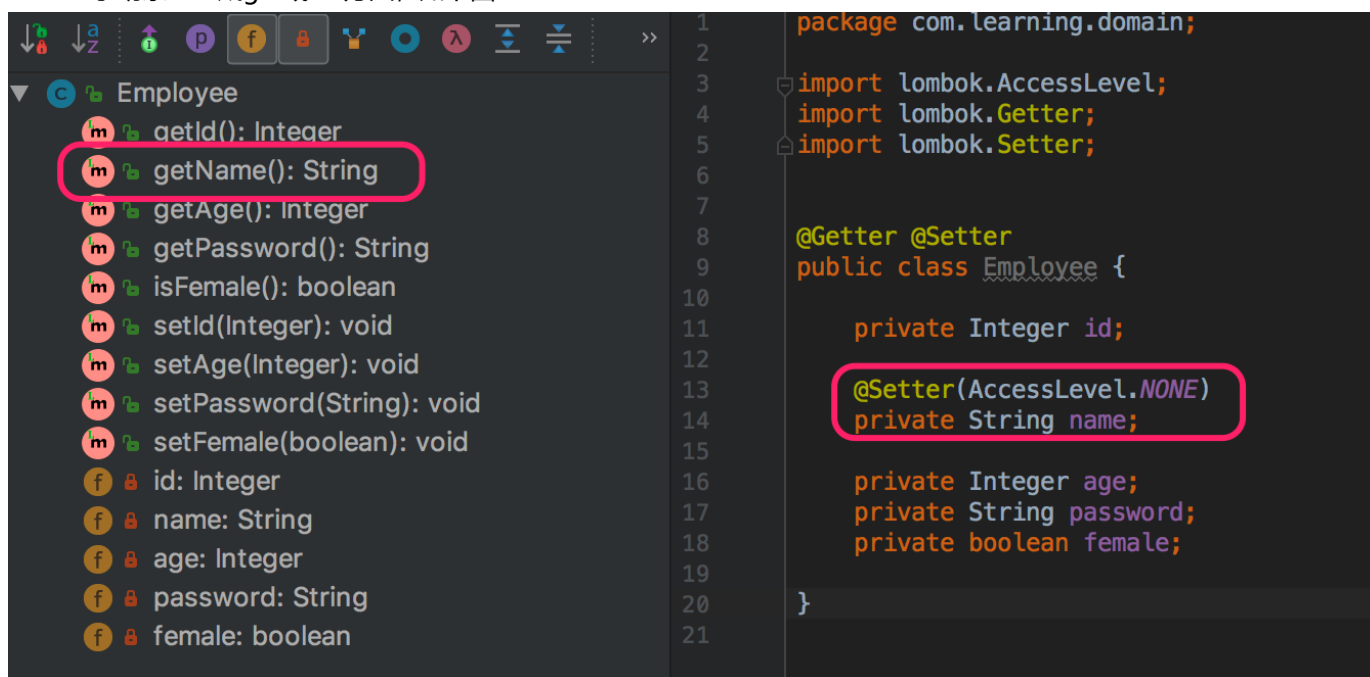


@Getter和@Setter

该注解可应用在类或成员变量之上，和我们预想的一样，`@Getter` 和 `@Setter` 就是为成员变量自动生成 `get` 和 `set` 方法，默认生成访问权限为 `public` 方法，当然我们也可以指定访问权限 `protected` 等，如下图：

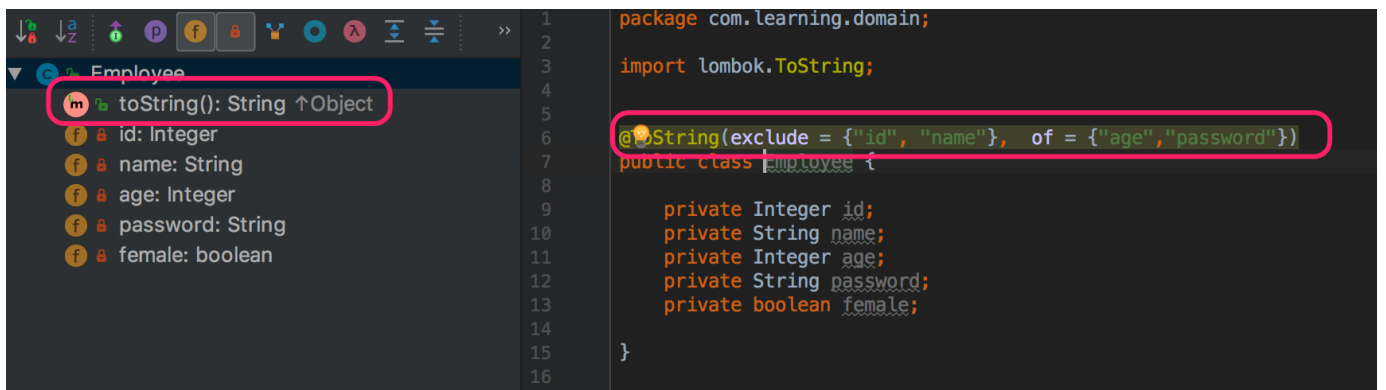


成员变量 `name` 指定生成 `set` 方法，并且访问权限为 `protected`；`boolean` 类型的成员变量 `female` 只生成 `get` 方法，并修改方法名称为 `isFemale()`。当把该注解应用在类上，默认为所有非静态成员变量生成 `get` 和 `set` 方法，也可以通过 `AccessLevel.NONE` 手动禁止生成 `get` 或 `set` 方法，如下图：

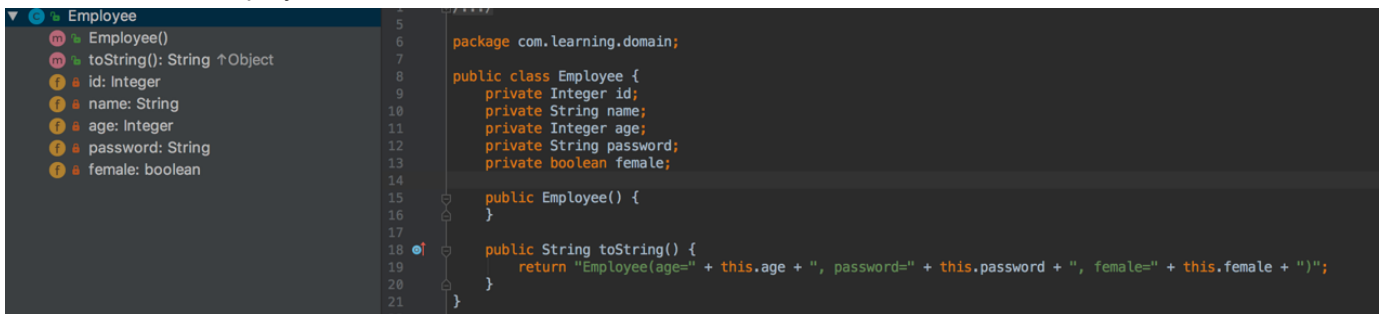


@ToString

该注解需应用在类上，为我们生成 `Object` 的 `toString` 方法，而该注解里面的几个属性能更加丰富我们想要的内容，`exclude` 属性禁止在 `toString` 方法中使用某字段，而 `of` 属性可以指定需要使用的字段，如下图：

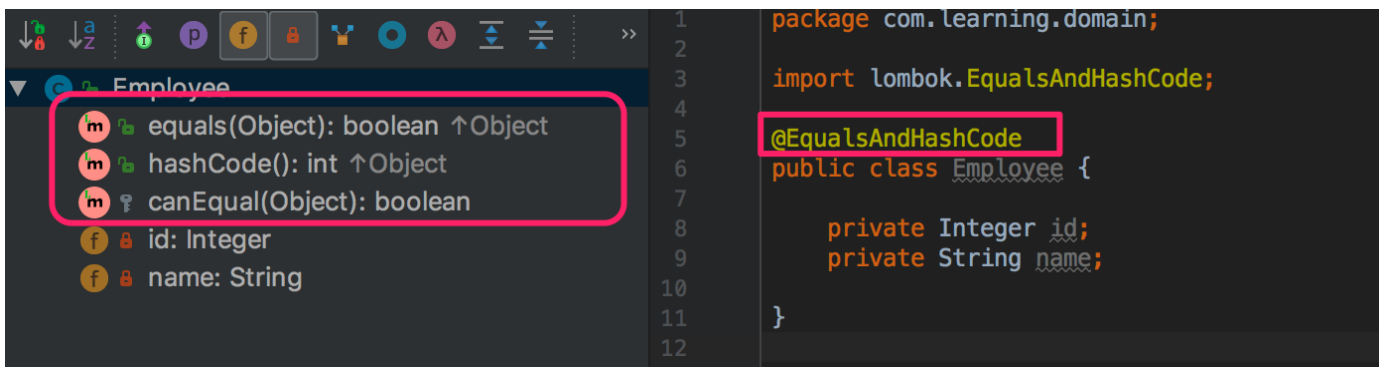


查看编译后的Employee.class得到我们预期的结果，如下图

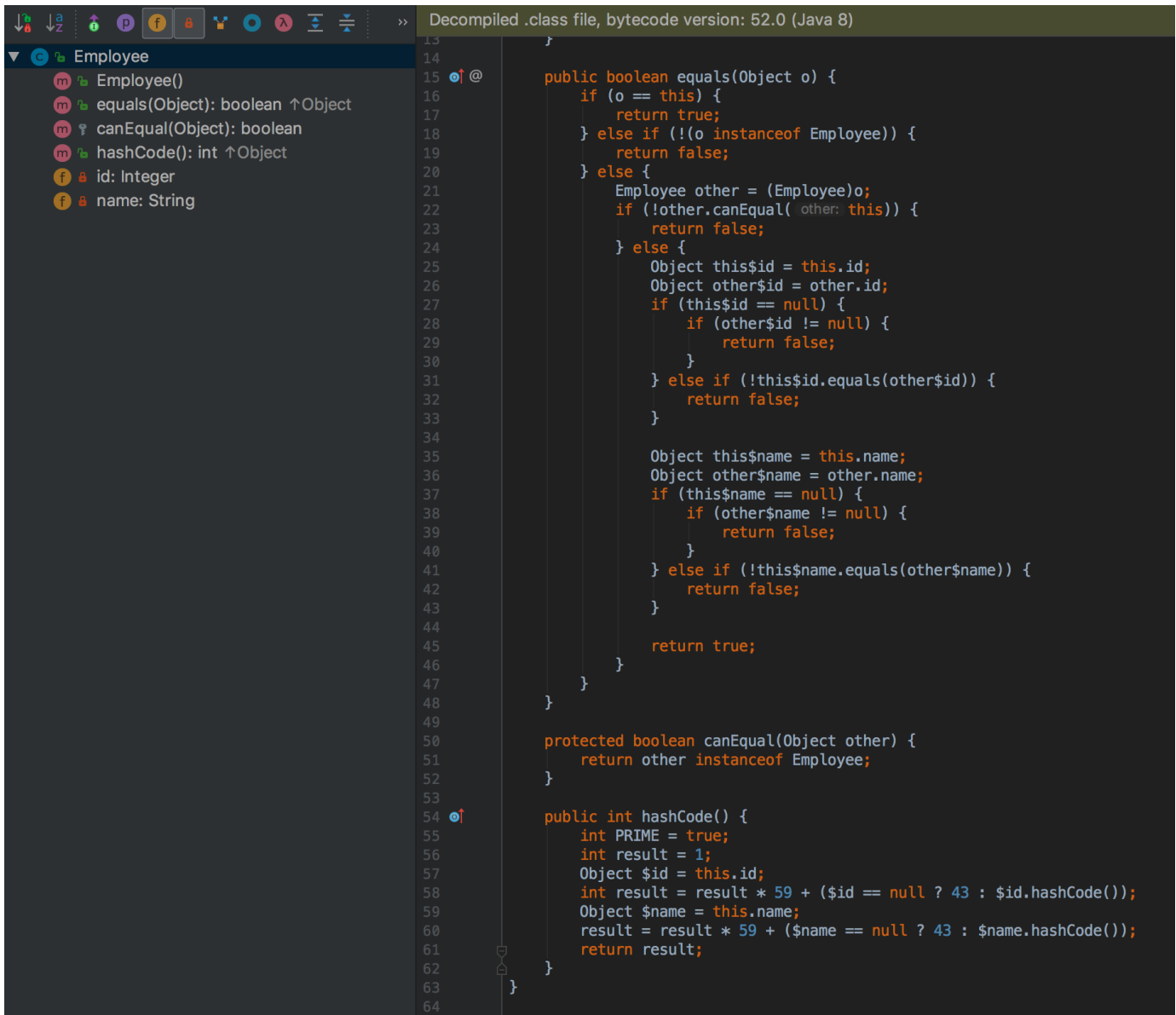


@EqualsAndHashCode

该注解需应用在类上，使用该注解，lombok会为我们生成 `equals(Object other)` 和 `hashCode()` 方法，包括所有非静态属性和非transient的属性，同样该注解也可以通过 `exclude` 属性排除某些字段，`of` 属性指定某些字段，也可以通过 `callSuper` 属性在重写的方法中使用父类的字段，这样我们可以更灵活的定义bean的比对，如下图：



查看编译后的Employee.class文件，如下图：

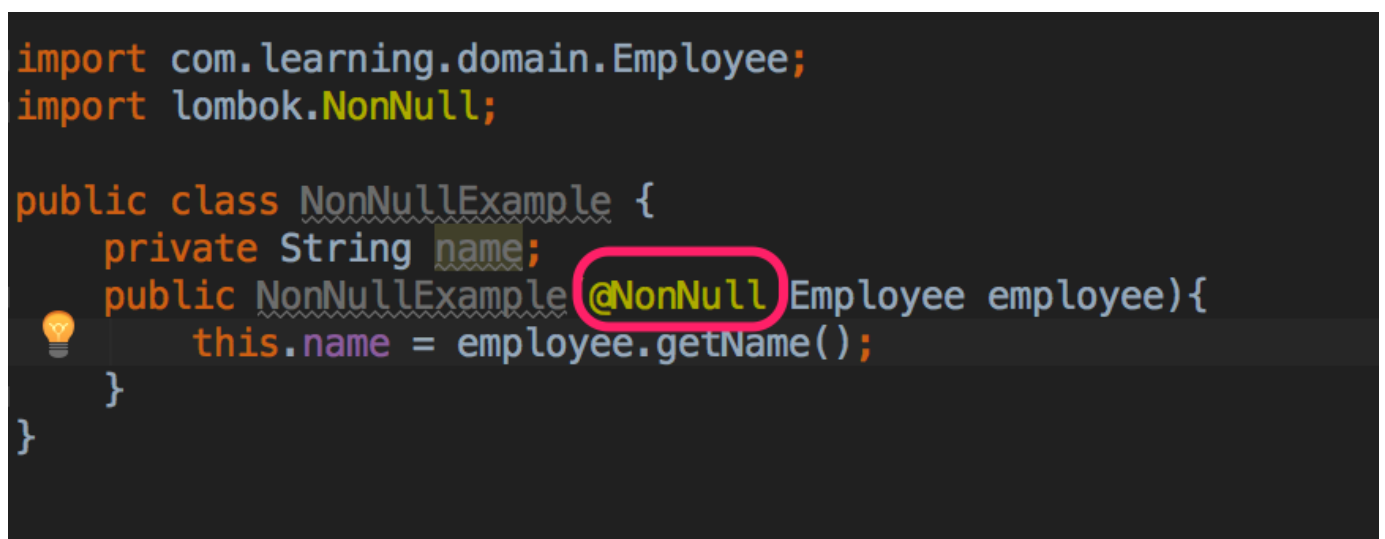


```
Decompiled .class file, bytecode version: 52.0 (Java 8)

13
14
15 @
16 public boolean equals(Object o) {
17     if (o == this) {
18         return true;
19     } else if (!(o instanceof Employee)) {
20         return false;
21     } else {
22         Employee other = (Employee)o;
23         if (!other.canEqual( other: this)) {
24             return false;
25         } else {
26             Object this$id = this.id;
27             Object other$id = other.id;
28             if (this$id == null) {
29                 if (other$id != null) {
30                     return false;
31                 }
32             } else if (!this$id.equals(other$id)) {
33                 return false;
34             }
35
36             Object this$name = this.name;
37             Object other$name = other.name;
38             if (this$name == null) {
39                 if (other$name != null) {
40                     return false;
41                 }
42             } else if (!this$name.equals(other$name)) {
43                 return false;
44             }
45
46             return true;
47         }
48     }
49
50 protected boolean canEqual(Object other) {
51     return other instanceof Employee;
52 }
53
54 public int hashCode() {
55     int PRIME = true;
56     int result = 1;
57     Object $id = this.id;
58     int result = result * 59 + ($id == null ? 43 : $id.hashCode());
59     Object $name = this.name;
60     result = result * 59 + ($name == null ? 43 : $name.hashCode());
61     return result;
62 }
63
64 }
```

@NonNull

该注解需应用在方法或构造器的参数上或属性上，用来判断参数的合法性，默认抛出 NullPointerException 异常



```
import com.learning.domain.Employee;
import lombok.NonNull;

public class NonNullExample {
    private String name;
    public NonNullExample(@NonNull Employee employee){
        this.name = employee.getName();
    }
}
```

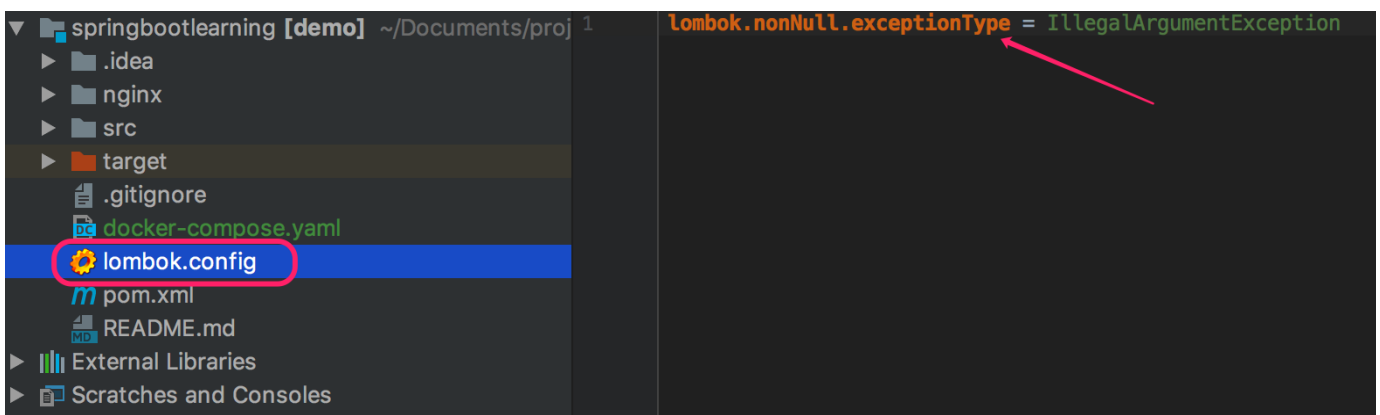
查看NonNullExample.class文件，会为我们抛出空指针异常，如下图：


```
import ...

public class NonNullExample {
    private String name;

    public NonNullExample(@NonNull Employee employee) {
        if (employee == null) {
            throw new NullPointerException("employee");
        } else {
            this.name = employee.getName();
        }
    }
}
```

当然我们可以通过指定异常类型抛出其他异常，`lombok.nonNull.exceptionType = [NullPointerException | IllegalArgumentException]`，为实现此功能我们需要在项目的根目录新建`lombok.config`文件：



重新编译NonNullExample类，已经为我们抛出非法参数异常：

```
package com.learning.service;

import ...

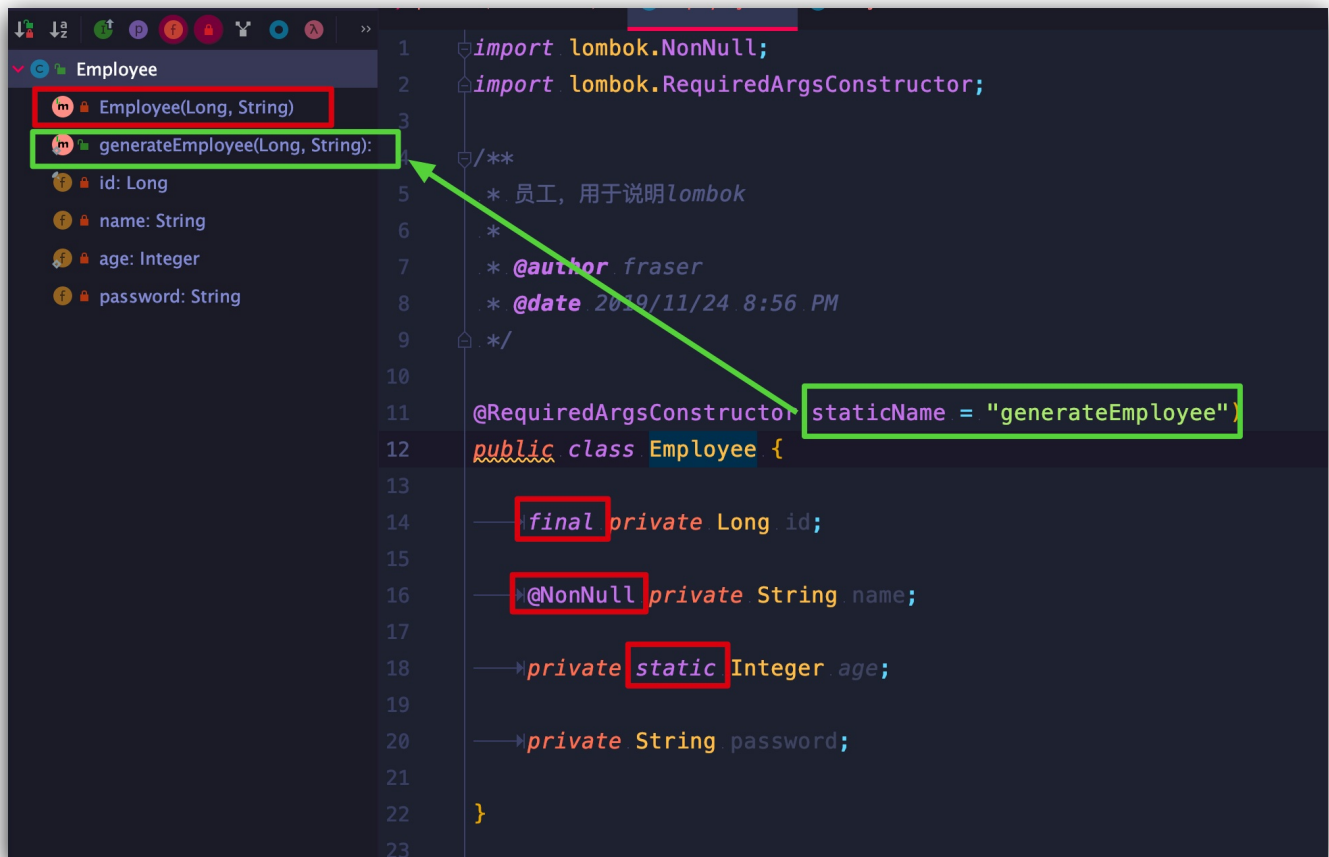
public class NonNullExample {
    private String name;

    public NonNullExample(@NonNull Employee employee) {
        if (employee == null) {
            throw new IllegalArgumentException("employee is null");
        } else {
            this.name = employee.getName();
        }
    }
}
```

@NoArgsConstructor, @RequiredArgsConstructor, @AllArgsConstructor

以上三个注解分别为我们生成无参构造器，指定参数构造器和包含所有参数的构造器，默认情况下，@RequiredArgsConstructor, @AllArgsConstructor 生成的构造器会对所有标记 @NonNull 的属性做非空校验。

无参构造器很好理解，我们主要看看后两种，先看 @RequiredArgsConstructor



从上图中我们可以看出，@RequiredArgsConstructor 注解生成有参数构造器时只会包含有 final 和 @NonNull 标识的 field，同时我们可以指定 staticName 通过生成静态方法来构造对象

查看Employee.class文件

```
public class Employee {
    private final Long id;
    @NonNull
    private String name;
    private static Integer age;
    private String password;

    private Employee(Long id, @NonNull String name) {
        if (name == null) {
            throw new NullPointerException("name is marked non-null but is null");
        } else {
            this.id = id;
            this.name = name;
        }
    }

    public static Employee generateEmployee(Long id, @NonNull String name) { return new Employee(id, name); }
}
```

当我们把 staticName 属性去掉我们来看遍以后的文件:

```
public class Employee {
    private final Long id;
    @NonNull
    private String name;
    private static Integer age;
    private String password;

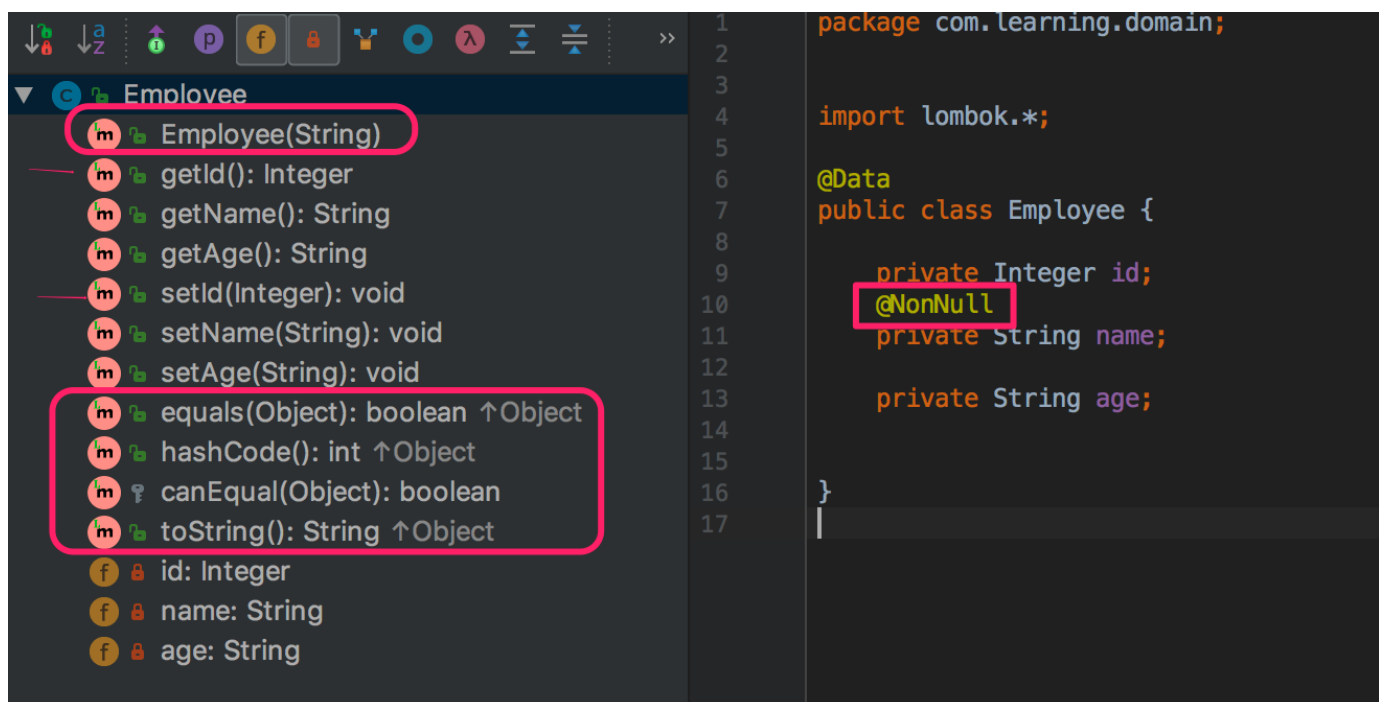
    public Employee(Long id, @NonNull String name) {
        if (name == null) {
            throw new NullPointerException("name is marked non-null but is null");
        } else {
            this.id = id;
            this.name = name;
        }
    }
}
```

相信你已经注意到细节

`@AllArgsConstructor` 就更简单了, 请大家自行查看吧

@Data

介绍了以上的注解, 再来介绍 `@Data` 就非常容易懂了, `@Data` 注解应用在类上, 是 `@ToString`, `@EqualsAndHashCode`, `@Getter` / `@Setter` 和 `@RequiredArgsConstructor` 合力的体现, 如下图:



```
package com.learning.domain;

import lombok.*;

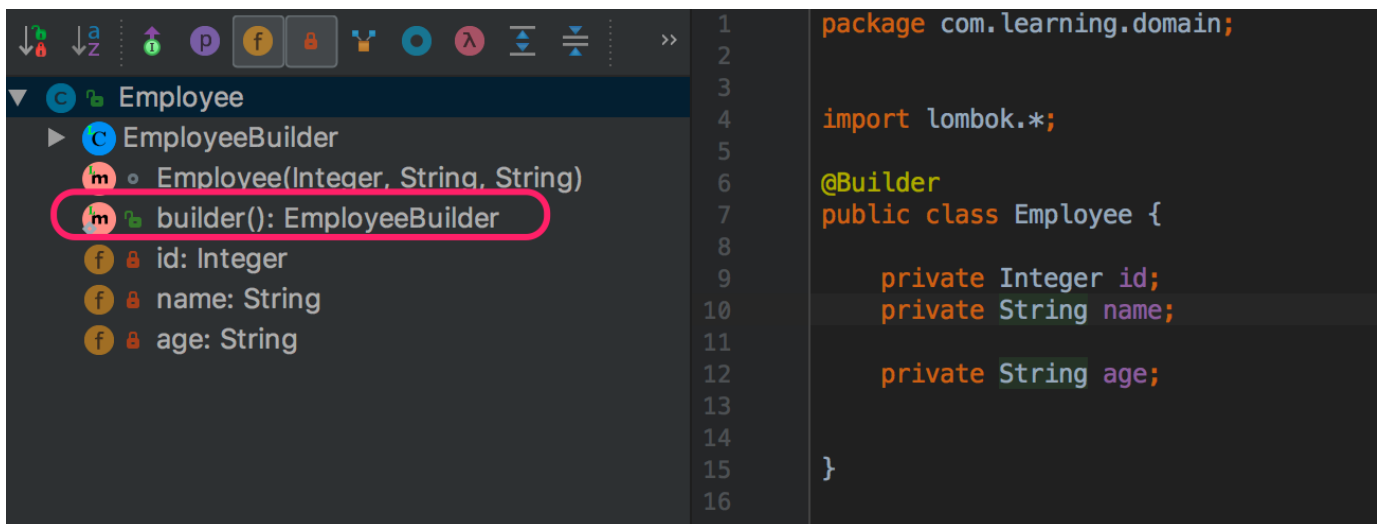
@Data
public class Employee {

    private Integer id;
    @NonNull
    private String name;

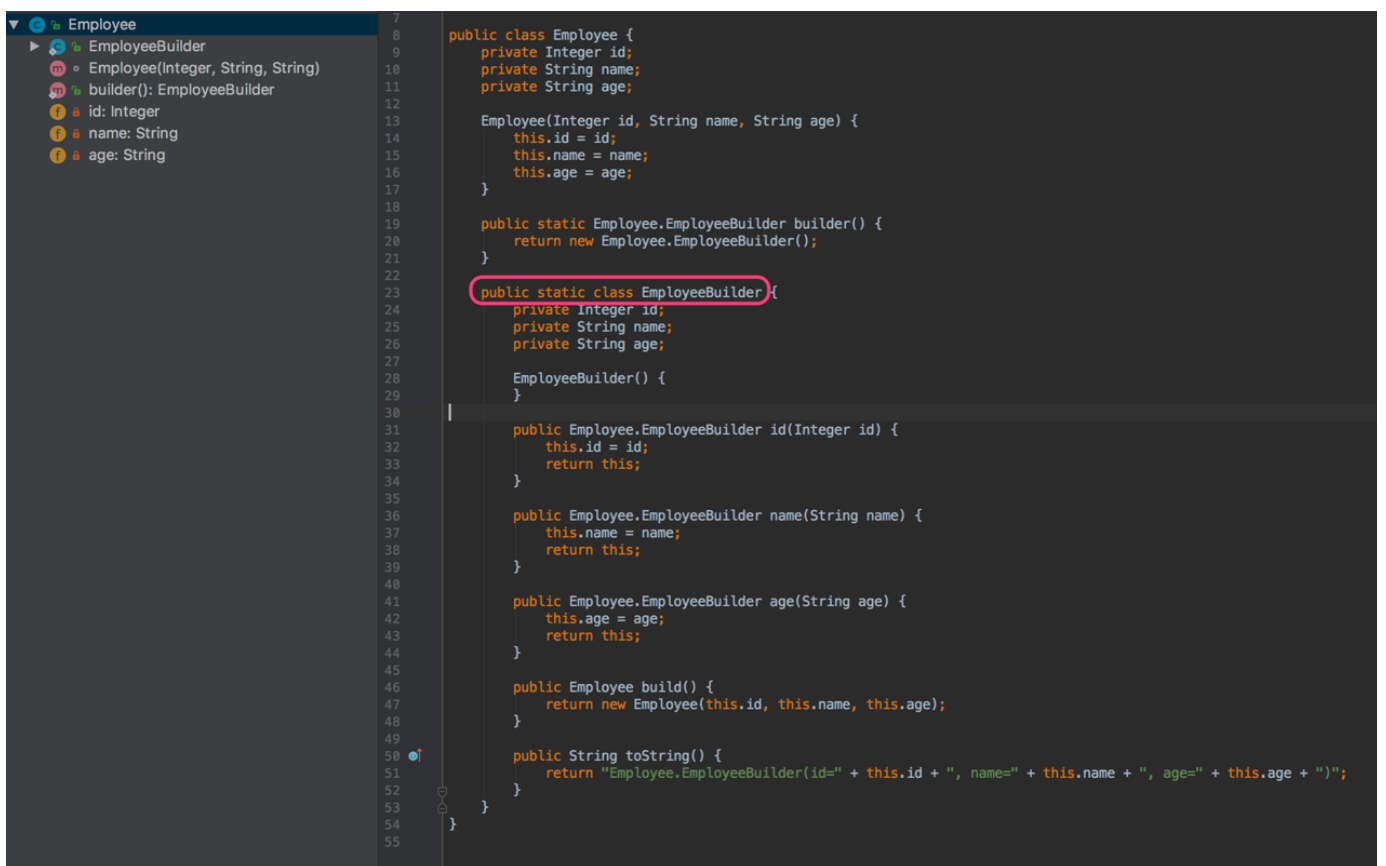
    private String age;
}
```

@Builder

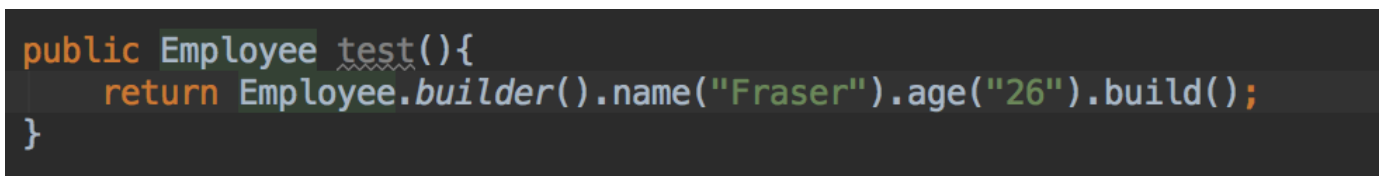
函数式编程或者说流式的操作越来越流行, 应用在大多数语言中, 让程序更具更简介, 可读性更高, 编写更连贯, `@Builder` 就带来了这个功能, 生成一系列的builder API, 该注解也需要应用在类上, 看下面的例子就会更加清晰明了。



编译后的Employee.class文件如下：



妈妈再也不用担心我 set 值那么麻烦了，流式操作搞定：



@Log

该注解需要应用到类上，在编写服务层，需要添加一些日志，以便定位问题，我们通常会定义一个静态常量Logger，然后应用到我们想日志的地方，现在一个注解就可以实现：

```

@Builder
@Log
public class Employee {

    private Integer id;
    private String name;

    private String age;

    public Employee test(){
        log.info( msg: "this is test method");
        return Employee.builder().name("Fraser").age("26").build();
    }
}

```

查看class文件，和我们预想的一样：

```

import java.util.logging.Logger;

public class Employee {
    private static final Logger log = Logger.getLogger(Employee.class.getName());
    private Integer id;
    private String name;
    private String age;

    public Employee test() {
        log.info( msg: "this is test method");
        return builder().name("Fraser").age("26").build();
    }
}

```

Log有很多变种，CommonLog，Log4j，Log4j2，Slf4j等，lombok依旧良好的通过变种注解做良好的支持：

```

@CommonsLog
Creates
private static final org.apache.commons.logging.Log log = org.apache.commons.logging.LogFactory.getLog(LogExample.class);
@JBossLog
Creates
private static final org.jboss.logging.Logger log = org.jboss.logging.Logger.getLogger(LogExample.class);
@Log
Creates
private static final java.util.logging.Logger log = java.util.logging.Logger.getLogger(LogExample.class.getName());
@Log4j
Creates
private static final org.apache.log4j.Logger log = org.apache.log4j.Logger.getLogger(LogExample.class);
@Log4j2
Creates
private static final org.apache.logging.log4j.Logger log = org.apache.logging.log4j.LogManager.getLogger(LogExample.class);
@Slf4j
Creates
private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(LogExample.class);
@XSlf4j
Creates
private static final org.slf4j.ext.XLogger log = org.slf4j.ext.XLoggerFactory.getXLogger(LogExample.class);

```

我实际使用的是 @Slf4j 注解

val

熟悉 Javascript 的同学都知道，var 可以定义任何类型的变量，而在 java 的实现中我们需要指定具体变量的类型，而 val 让我们摆脱指定，编译之后就精准匹配上类型，默认是 final 类型，就像 java8 的函数式表达式，() \rightarrow System.out.println("hello lombok"); 就可以解析到Runnable函数式接口。

```
import lombok.*;

import java.util.HashMap;

public class Employee {

    private Integer id;
    private String name;

    private String age;

    public void test(){
        val map = new HashMap<String,String>();
        map.put("name","Fraser");
        map.put("age","26");
    }

}
```

查看解析后的class文件：

```
import java.util.HashMap;

public class Employee {
    private Integer id;
    private String name;
    private String age;

    public Employee() {
    }

    public void test() {
        HashMap<String, String> map = new HashMap();
        map.put("name", "Fraser");
        map.put("age", "26");
    }
}
```

@Cleanup

当我们对流进行操作，我们通常需要调用 close 方法来关闭或结束某资源，而 @Cleanup 注解可以帮助我们调用 close 方法，并且放到 try/finally 处理块中，如下图：

```
import lombok.Cleanup;
import java.io.*;

public class NonNullExample {

    public static void main(String[] args) throws IOException {
        @Cleanup InputStream in = new FileInputStream(args[0]);
        @Cleanup OutputStream out = new FileOutputStream(args[1]);
        byte[] b = new byte[10000];
        while (true) {
            int r = in.read(b);
            if (r == -1) break;
            out.write(b, 0, r);
        }
    }
}
```


编译后的class文件如下，我们发现被try/finally包围处理，并调用了流的close方法

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Collections;

public class NonNullExample {
    public NonNullExample() {
    }

    public static void main(String[] args) throws IOException {
        FileInputStream in = new FileInputStream(args[0]);
        try {
            FileOutputStream out = new FileOutputStream(args[1]);
            try {
                byte[] b = new byte[10000];

                while(true) {
                    int r = in.read(b);
                    if (r == -1) {
                        return;
                    }

                    out.write(b, off: 0, r);
                }
            } finally {
                if (Collections.singletonList(out).get(0) != null) {
                    out.close();
                }
            }
        } finally {
            if (Collections.singletonList(in).get(0) != null) {
                in.close();
            }
        }
    }
}
```

其实在 JDK1.7 之后就有了 `try-with-resource`，不用我们显式的关闭流，这个请大家自行看吧

总结

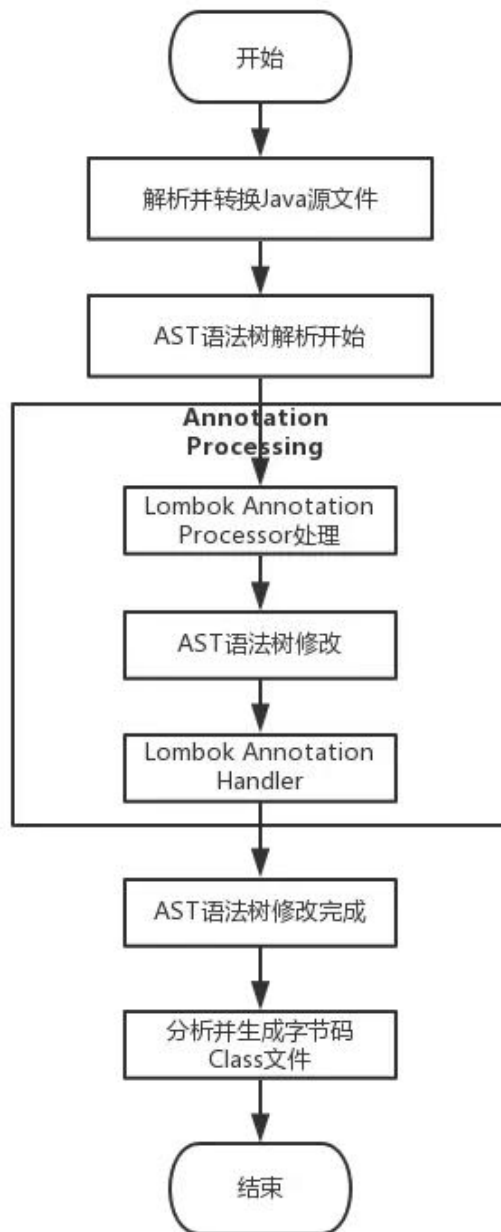
Lombok的基本操作流程是这样的：

定义编译期的注解

利用JSR269 api(Pluggable Annotation Processing API)创建编译期的注解处理器

利用tools.jar的javac api处理AST(抽象语法树)

将功能注册进jar包



Lombok 当然还有很多注解，我推荐使用以上就足够了，这个工具是带来便利的，而不能被其捆绑，“弱水三千只取一瓢饮，代码千万需抓重点看”，Lombok 能让我更加专注有效代码排除意义微小的障眼代码（get, set等），另外Lombok生成的代码还能像使用工具类一样方便（@Builder）。

更多内容请查看官网：<https://www.projectlombok.org/>